Convolution Coder Software Implementation Using VIiterbiDecoding Algorithm

Othman O. Khalifa, Tariq Al-maznaee
Department of Electrical and Compute Engineering
International Islamic University Malaysia
53100 Kuala Lumpur, Malaysia
e-mail: khalifa@iiu.edu.my

Mahmood Munjid and Aisha Hussan thors Name/s per
Department of Electrical and Compute Engineering
International Islamic University Malaysia
53100 Kuala Lumpur, Malaysia
e-mail: khalifa@iiu.edu.my

Abstract—**This In this paper, Convolutional coder software implementation using Viterbi decoding algorithm for bitstream that has been encoded using Forward error correction is presented. It will discuss the detailed description and the steps involved in simulating a communication channel using convolutional encoding with Viterbi decoding. The steps will involve generating random binary data, convolutionally encoding the data, passing the encoded data through a noisy channel, quantizing the received channel symbols, and finally performing Viterbi decoding on the quantized channel symbols to recover the original binary data. The paper will be in a convincing manner and authors aim to explain to the reader the advantages of convolutional coding with Viterbi decoding over conventional decoding techniques in terms of BER.**

Keywords-component; Forward error correction, Convolution coding, Viterbi decoding algorithm

I. INTRODUCTION

The purpose of this paper is to implement software that convolutionally code bitstream in the transmitter side and then decode that bitstream in the receiver side using Viterbi decoding algorithm on the bases of forward error correction technique. The purpose of Forward Error Correction (FEC) is to improve the capacity of a channel by adding some carefully designed redundant information to the data being transmitted. The process of adding this redundant information is known as *channel coding*. More particularly, this project will focus primarily on the Viterbi decoding algorithm itself. Viterbi decoding was developed by Andrew J. Viterbi, a founder of Qualcomm Corporation [1]. Since then, other researchers have expanded on his work by finding good convolutional codes, exploring the performance limits of the technique, and varying decoder design parameters to optimize the implementation of the technique in hardware and software. The Viterbi decoding algorithm is also used in decoding trellis-coded modulation, invented by Gottfried Ungerboeck in 1982. This is the technique used in telephone-line modems to squeeze high ratios of bits-per-second to Hertz out of 3 kHz-bandwidth analog telephone lines [5]. Viterbi decoding is one of two types of decoding algorithms used with convolutional encoding the other type is *sequential decoding*. Sequential decoding has the advantage that it can perform very well with long-constraint-length convolutional codes, but it has a variable decoding time [2]. The discussion of sequential decoding algorithms will not be highlighted here since it is beyond the scope of this project.

In this paper will go in detailed description of the algorithms for generating random binary data, convolutionally encoding the data, passing the encoded data through a noisy channel, quantizing the received channel symbols, and finally performing Viterbi decoding on the quantized channel symbols to recover the original binary data.

Convolutional coding and block coding are the two major forms of channel coding. Convolutional codes operate on serial data, one or a few bits at a time. Block codes operate on relatively large (typically, up to a couple of hundred bytes) message blocks. There are a variety of useful convolutional and block codes, and a variety of algorithms for decoding the received coded information sequences to recover the original data [5, 7].

Convolutional encoding using Viterbi decoding is a FEC technique that is particularly suited to a channel in which the transmitted signal is corrupted mainly by additive white gaussian noise (AWGN). You can think of AWGN as noise whose voltage distribution over time has characteristics that can be described using a Gaussian, or normal, statistical distribution, i.e. a bell curve. This voltage distribution has zero mean and a standard deviation that is a function of the signal-to-noise ratio (SNR) of the received signal. Let's assume for the moment that the received signal level is fixed. Then if the SNR is high, the standard deviation of the noise is small, and vice-versa. In digital communications, SNR is usually measured in terms of E_b/N_0, which stands for energy per bit divided by the one-sided noise density [3].

Let's take a moment to look at a couple of examples. Suppose that we have a system where a '1' channel bit is transmitted as a voltage of -1V, and a '0' channel bit is transmitted as a voltage of +1V. This is called bipolar non-return-to-zero (bipolar NRZ) signaling. It is also called binary "antipodal" (which means the signaling states are exact opposites of each other) signaling. The receiver comprises a comparator that decides the received channel bit is a '1' if its voltage is less than 0V, and a '0' if its voltage is greater than or

equal to 0V. One would want to sample the output of the comparator in the middle of each data bit interval. Let's see how our example system performs, first, when the E_b/N_0 is high, and then when the E_b/N_0 is lower.

Figure-1 shows the results of a channel simulation where one million (1 x 10^6) channel bits are transmitted through an AWGN channel with an E_b/N_0 level of 20 dB (i.e. the signal voltage is ten times the rms noise voltage). In this simulation, a '1' channel bit is transmitted at a level of -1V, and a '0' channel bit is transmitted at a level of +1V. The x-axis of this figure corresponds to the received signal voltages, and the y-axis represents the number of times each voltage level was received.

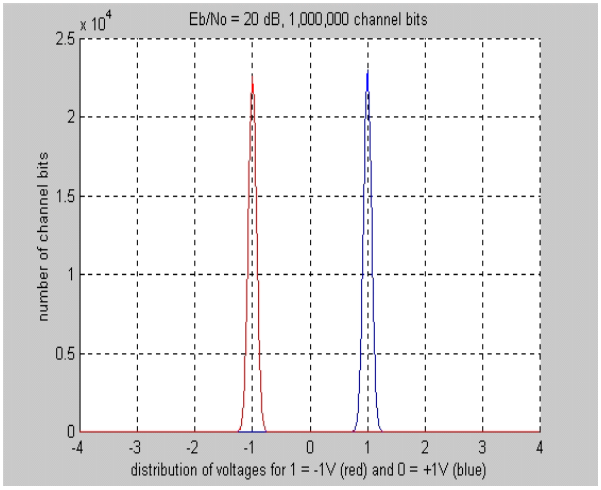

Figure 1:Tthe results of a channel simulation where one million (1 x 10^6) channel bits are transmitted through an AWGN channel with an E_b/N_0 level of 20 dB.

Our simple receiver detects a received channel bit as a '1' if its voltage is less than 0V, and as a '0' if its voltage is greater than or equal to 0V. Such a receiver would have little difficulty correctly receiving a signal as depicted in the Figure-1. Very few (if any) channel bit reception errors would occur. In this example simulation with the E_b/N_0 set at 20 dB, a transmitted '0' was never received as a '1', and a transmitted '1' was never received as a '0'. So far, so good.

In Figure-2 the results of a similar channel simulation was conducted when one million (1 x 10^6) channel bits are transmitted through an AWGN channel where the E_b/N_0 level has decreased to 6 dB (i.e. the signal voltage is two times the rms noise voltage).

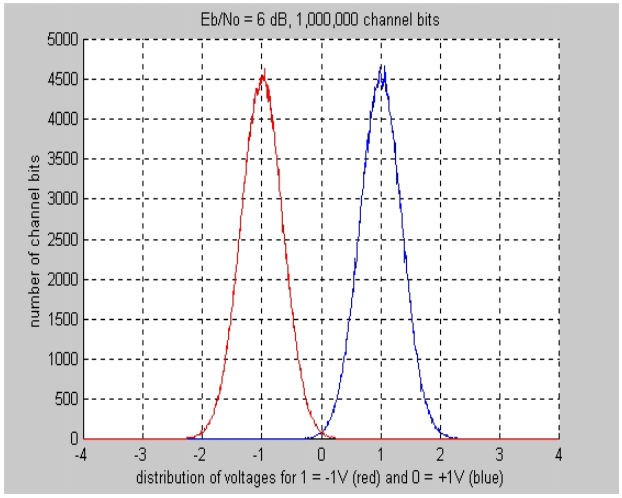

Figure 1:Tthe results of a channel simulation when one million (1 x 10^6) channel bits are transmitted through an AWGN channel where the E_b/N_0 level has decreased to 6 dB.

Now observe how the right-hand side of the red curve in the figure above crosses 0V, and how the left-hand side of the blue curve also crosses 0V.

The points on the red curve that are above 0V represent events where a channel bit that was transmitted as a one (-1V) was received as a zero. The points on the blue curve that are below 0V represent events where a channel bit that was transmitted as a zero (+1V) was received as a one. Obviously, these events correspond to channel bit reception errors in our simple receiver. In this example simulation with the E_b/N_0 set at 6 dB, a transmitted '0' was received as a '1' 1,147 times, and a transmitted '1' was received as a '0' 1,207 times, corresponding to a bit error rate (BER) of about 0.235%.

That's not so good, especially if you're trying to transmit highly compressed data, such as digital television [4].

In this paper we will show that convolutional coding using Viterbi decoding, allows achieving BER of better than 1 x 10^{-7} at the same E_b/N_0, 6 dB.

Convolutional codes are usually described using two parameters: the *code rate* (R) and the *constraint length* (L). The code rate, R= k/n, is expressed as a ratio of the number of bits into the convolutional encoder (k) to the number of channel symbols output by the convolutional encoder (n) in a given encoder cycle. The constraint length parameter, L, denotes the "length" of the convolutional encoder, i.e. how many k-bit stages are available to feed the combinatorial logic that produces the output symbols. Closely related to L is the parameter m, which indicates how many encoder cycles an input bit is retained and used for encoding after it first appears at the input to the convolutional encoder. The m parameter can be thought of as the memory length of the encoder [1]. In this work we focused on rate 1/2 convolutional codes.

II. DESCRIPTION OF THE ALGORITHMS

The steps involved in simulating a communication channel using convolutional encoding with Viterbi decoding are as follows:

- Generate the data to be transmitted through the channel-result is binary data bits
- Convolutionally encode the data in to channel symbols
- Map the one/zero channel symbols into an antipodal baseband signal, producing transmitted channel symbols
- Add noise to the transmitted channel symbols-result is received channel symbols
- Quantize the received channel levels-one bit quantization called hard-decision, and two to n bit quantization is called soft-decision (n is usually three or four)
- Perform Viterbi decoding on the quantized received channel symbols-result is again binary data bits
- Compare the decoded data bits to the transmitted data bits and count the number of errors.

You may notice that we left out the steps of modulating the channel symbols onto a transmitted carrier, and then demodulating the received carrier to recover the channel symbols. This is because we can accurately model the effects of AWGN even though we bypass those steps [3].

A. Generating the Data

Generating the data to be transmitted through the channel can be accomplished quite simply by using a random number generator. A uniform distribution of numbers on the interval between 0 to a maximum value of one, we can say that any value less than half of the maximum value is a zero; any value greater than or equal to half of the maximum value is a one.

B. Convolutionally Encoding the Data

Convolutionally encoding the data is accomplished using a shift register and associated combinatorial logic that performs modulo-two addition. (A shift register is merely a chain of flip-flops wherein the output of the nth flip-flop is tied to the input of the (n+1)th flip-flop. Every time the active edge of the clock occurs, the input to the flip-flop is clocked through to the output, and thus the data are shifted over one stage.) The combinatorial logic is often in the form of cascaded exclusive-or gates. As a reminder, exclusive-or gates are two-input, one-output gates often represented by the logic symbol shown in figure-3 with truth-table shown below.

Figure 3: A logical symbol of exclusive-or gate [6].

Table 1: Exclusive-or gate truth-table [6].

Input A	Input B	Output (A xor B)
0	0	0
0	1	1
1	0	1
1	1	0

The exclusive-or gate performs modulo-two addition of its inputs. When you cascade q two-input exclusive-or gates, with the output of the first one feeding one of the inputs of the second one, the output of the second one feeding one of the inputs of the third one, etc., the output of the last one in the chain is the modulo-two sum of the q + 1 inputs. Another way to illustrate the modulo-two adder, and the way that is most commonly used in textbooks, is as a circle with a + symbol inside, thus: $\oplus$

Now that we have the two basic components of the convolutional encoder (flip-flops comprising the shift register and exclusive-or gates comprising the associated modulo-two adders) defined, let's look at figure-4 of a convolutional encoder of R= 1/2, K = 3, m = 2 codes:

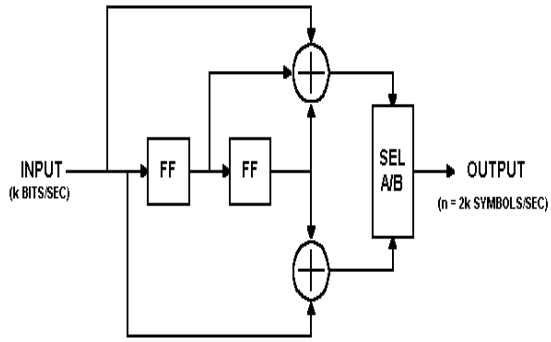

Figure 4: Coder of R= 1/2, K =3, m =2 code.

In this encoder, data bits are provided at a rate of k-bits per second. Channel symbols are output at a rate of n = 2k symbols per second. The input bit is stable during the encoder cycle. The encoder cycle starts when an input clock edge occurs. When the input clock edge occurs, the output of the left-hand flip-flop is clocked into the right-hand flip-flop, the previous input bit is clocked into the left-hand flip-flop, and a new input bit becomes available. Then the outputs of the upper and lower modulo-two adders become stable. The output selector (SEL A/B block) cycles through two states-in the first state, it selects and outputs the output of the upper modulo-two adder; in the second state, it selects and outputs the output of the lower modulo-two adder.

The encoder shown in figure-4 encodes the K = 3, (7, 5) convolutional code. The octal numbers 7 and 5 represent the code generator polynomials, which when read in binary (111_2 and 101_2) correspond to the shift register connections to the upper and lower modulo-two adders, respectively. This code has been determined to be the "best" code for rate 1/2, K = 3. It

is the code we will use for the remaining discussion, for reasons that will become readily apparent when we get into the Viterbi decoder algorithm.

Now let's consider the following input data stream, and the corresponding output data stream:

Let the input sequence date, d= 010111001010001_2 and the outputs of both of the flip-flops in the shift register are initially cleared, i.e. their outputs are zeroes. The first clock cycle makes the first input bit, a zero, available to the encoder. The flip-flop outputs are both zeroes. The inputs to the modulo-two adders are all zeroes, so the output of the encoder is 00_2.

The second clock cycle makes the second input bit available to the encoder. The left-hand flip-flop clocks in the previous bit, which was a zero, and the right-hand flip-flop clocks in the zero output by the left-hand flip-flop. The inputs to the top modulo-two adder are 100_2, so the output is a one. The inputs to the bottom modulo-two adder are 10_2, so the output is also a one. So the encoder outputs 11_2 for the channel symbols.

The third clock cycle makes the third input bit, a zero, available to the encoder. The left-hand flip-flop clocks in the previous bit, which was a one, and the right-hand flip-flop clocks in the zero from two bit-times ago. The inputs to the top modulo-two adder are 010_2, so the output is a one. The inputs to the bottom modulo-two adder are 00_2, so the output is zero. So the encoder outputs 10_2 for the channel symbols. And so on. The timing diagram that illustrates the process is shown in figure-5.

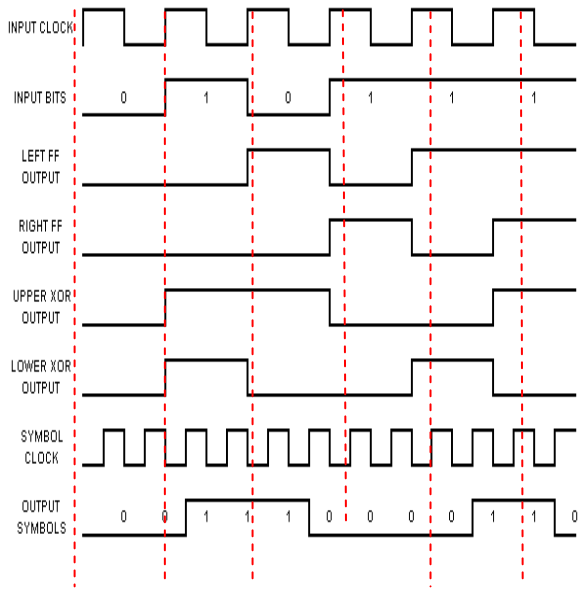

Figure 5: The timing diagram of the encoder for the first 6-bits.

After all of the inputs have been presented to the encoder, the output sequence will be:

$00\ 11\ 10\ 00\ 01\ 10\ 01\ 11\ 11\ 10\ 00\ 10\ 11\ 00\ 11_2$.

Notice here that we have paired the encoder outputs-the first bit in each pair is the output of the upper modulo-two adder; the second bit in each pair is the output of the lower modulo-two adder.

You can see from the structure of the R = 1/2, K =3 convolutional encoder and from the case given above that each input bit has an effect on three successive pairs of output symbols. That is an extremely important point and that is what gives the convolutional code its error-correcting power. The reason why will become evident when we get into the Viterbi decoder algorithm.

Now if we are only going to send the 15 data bits given above, in order for the last bit to affect three pairs of output symbols, we need to output two more pairs of symbols. This is accomplished by clocking the convolutional encoder flip-flops two (= m) more times, while holding the input at zero.

This is called "flushing" the encoder, and results in two more pairs of output symbols. The final binary output of the encoder is thus:

O/P = $00\ 11\ 10\ 00\ 01\ 10\ 01\ 11\ 11\ 10\ 00\ 10\ 11\ 00\ 11\ 10\ 11_2$.

If we don't perform the flushing operation, the last m bits of the message have less error-correction capability than the first through (m - 1)th bits had. This is an important thing if we are going to use this FEC technique in a burst-mode environment. So's the step of clearing the shift register at the beginning of each burst. The encoder must start in a known state and end in a known state for the decoder to be able to reconstruct the input data sequence properly.

Now, let's look at the encoder from another perspective. We can think of the encoder as a simple state machine. The case above has two bits of memory, so there are four possible states. Let's give the left-hand flip-flop a binary weight of 2^1, and the right-hand flip-flop a binary weight of 2^0. Initially, the encoder is in the all-zeroes state. If the first input bit is a zero, the encoder stays in the all zeroes state at the next clock edge. But if the input bit is a one, the encoder transitions to the 10_2 state at the next clock edge. Then, if the next input bit is zero, the encoder transitions to the 01_2 state, otherwise, it transitions to the 11_2 state. The following table gives the next state given the current state and the input, with the states given in binary

Table 2: The next state table

	Next State, if	
Current State	**Input = 0:**	**Input = 1:**

00	00	10
01	00	10
10	01	11
11	01	11

Table-2 is often called a state transition table. We'll refer to it as the next state table.

Now let us look at a table that lists the channel output symbols, given the current state and the input data, which we'll refer to as the output table:

Table 3: The output table.

	Next State, if	
Current State	**Input = 0:**	**Input = 1:**
00	00	11
01	11	00
10	10	01
11	01	10

With these two tables, we can completely describe the behavior of the example rate 1/2, K = 3 convolutional encoder. Note that both of these tables have $2^{(K-1)}$ rows, and 2^k columns, where K is the constraint length and k is the number of bits input to the encoder for each cycle. These two tables will come in handy when we start discussing the Viterbi decoder algorithm.

C. *Mapping the Channel Symbols to Signal Levels*

Mapping the one/zero output of the convolutional encoder onto an antipodal baseband signaling scheme is simply a matter of translating zeroes to +1s and ones to -1s. This can be accomplished by performing the operation y = 1 - 2x on each convolutional encoder output symbol

D. *Adding Noise to the Transmitted Symbols*

Adding noise to the transmitted channel symbols produced by the convolutional encoder involves generating Gaussian random numbers, scaling the numbers according to the desired energy per symbol to noise density ratio, Es/N_0, and adding the scaled Gaussian random numbers to the channel symbol values.

For the uncoded channel, $Es/N_0 = E_b/N_0$, since there is one channel symbol per bit. However, for the coded channel, $Es/N_0 = E_b/N_0 + 10\log_{10}(k/n)$. For example, for rate 1/2 coding, $Es/N_0 = E_b/N_0 + 10\log_{10}(1/2) = E_b/N_0 - 3.01$ dB. Similarly, for rate 2/3 coding, $Es/N_0 = E_b/N_0 + 10\log_{10}(2/3) = E_b/N_0 - 1.76$ dB.

The Gaussian random number generator is the only interesting part of this task. C only provides a uniform random number generator using the function, *rand()*. In order to obtain Gaussian random numbers, we take advantage of relationships between uniform, Rayleigh, and Gaussian distributions:

Given a uniform random variable U, a Rayleigh random variable R can be obtained by:

$$R = \sqrt{2 \cdot \sigma^2 \cdot \ln(1/(1-U))} = \sigma \cdot \sqrt{2 \cdot \ln(1/(1-U))} \quad (1)$$

Where σ^2 is the variance of the Rayleigh random variable, and given R and a second uniform random variable V, two Gaussian random variables G and H can be obtained by:

$G = R \cos V;$

$H = R \sin V$ (2)

In the AWGN channel, the signal is corrupted by additive noise, n(t), which has the power spectrum *No*/2 watts/Hz. The variance σ^2 of this noise is equal to *No*/2. If we set the energy per symbol Es equal to 1, then

$$E_s/N_0 = 1/2\sigma^2 \quad (3)$$

Where;

$$\sigma = \sqrt{1/(2 \cdot (E_s/N_0))} \quad (4)$$

E. *Quantizing the Received Channel Symbols*

An ideal Viterbi decoder would work with infinite precision, or at least with floating-point numbers. In practical systems, we quantize the received channel symbols with one or a few bits of precision in order to reduce the complexity of the Viterbi decoder, not to mention the circuits that precede it. If the received channel symbols are quantized to one-bit precision (< 0V = 1, ≥ 0V = 0), the result is called hard-decision data. If the received channel symbols are quantized with more than one bit of precision, the result is called soft-decision data. A Viterbi decoder with soft decision data inputs quantized to three or four bits of precision can perform about 2 dB better than the one working with hard-decision inputs.

It is important to note that, usual quantization precision is three bits, where more than three bits provides little additional improvement.

The selection of the quantizing levels is an important design decision because it can have a significant effect on the performance of the link. The following is a very brief explanation of one way to set those levels.

Let's assume our received signal levels in the absence of noise are -1V = 1, +1V = 0. With noise, our received signal has mean +/-1 and standard deviation of:

$$\sigma = \sqrt{1/(2 \cdot (E_s/N_0))} \quad (5)$$

Let's use a uniform three-bit quantizer having the input/output relationship shown in figure-6, where D is a decision level that we will calculate shortly:

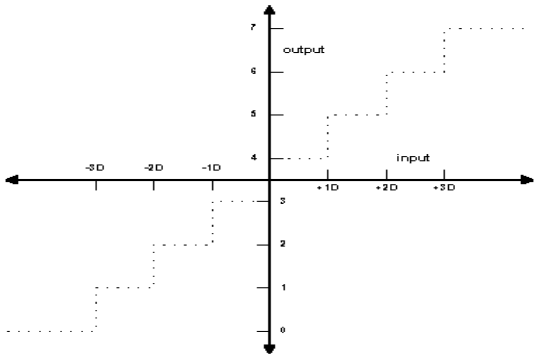

Figure 5: Input/Output relationship of uniform, three-bit quantizer.

The decision level, D, can be calculated according to the formula:

$$D = 0.5 \cdot \sigma = 0.5 \cdot \sqrt{1/(2 \cdot (E_s/N_0))} \qquad (6)$$

Where;

E_s/N_0 is the energy per symbol to noise density ratio [3].

F. Perform Viterbi decoding

The Viterbi decoder is the primary focus of this project. Perhaps the single most important concept to aid in understanding the Viterbi algorithm is the trellis diagram. Figure-5 below shows the trellis diagram for our example R= 1/2, K = 3 convolutional encoder for15-bit messages:

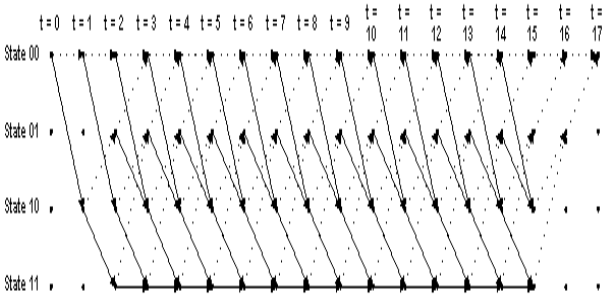

Figure 6: The trellis diagram for15-bits with two encoder memory flushing bits

The four possible states of the encoder are depicted as four rows of horizontal dots. There is one column of four dots for the initial state of the encoder and one for each time instant during the message.

For a 15-bit message with two encoder memory flushing bits, there are 17 time instants in addition to t = 0, which represents the initial condition of the encoder. The solid lines connecting dots in the diagram represent state transitions when the input bit is a one. The dotted lines represent state transitions when the input bit is a zero. Notice the correspondence between the arrows in the trellis diagram and the state transition table discussed above. Also notice that since the initial condition of the encoder is State 00_2, and the two memory flushing bits are zeroes, the arrows start out at State 00_2 and end up at the same state. The following diagram shows the states of the trellis that are actually reached during the encoding of our example 15-bit message:

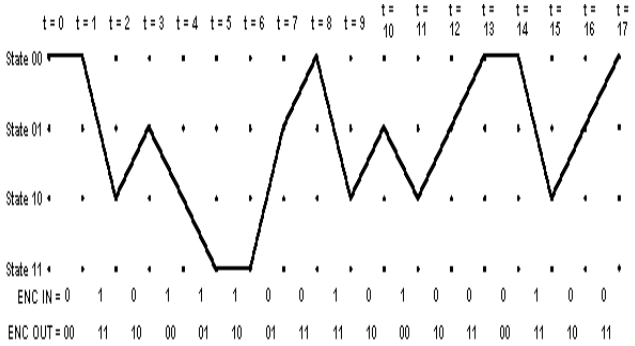

Figure 7: The trellis that are actually reached during the encoding of our example 15-bit message

The encoder input bits and output symbols are shown at the bottom of the diagram. Notice the correspondence between the encoder output symbols and the output table discussed above. Let's look at that in more detail, using the expanded version of the transition between one time instant to the next shown below:

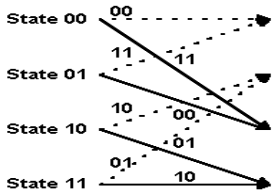

Figure 8: The expanded version of the transition between one time instant to the next

The two-bit numbers labeling the lines are the corresponding convolutional encoder channel symbol outputs. Remember that dotted lines represent cases where the encoder input is a zero, and solid lines represent cases where the encoder input is a one. (In figure-8, the two-bit binary numbers labeling dotted lines are on the left, and the two-bit binary numbers labeling solid lines are on the right.)

Now let's start looking at how the Viterbi decoding algorithm actually works. For our example, we're going to use hard-decision symbol inputs to keep things simple. (The example source code uses soft-decision inputs to achieve better performance.) Suppose we receive the above encoded message with a couple of bit errors:

Each time we receive a pair of channel symbols, we compute a metric to measure the "distance" between what we

received and all of the possible channel symbol pairs we could have received. Going from t = 0 to t = 1, there are only two possible channel symbol pairs we could have received: 00_2, and 11_2. That's because we know the convolutional encoder was initialized to the all-zeroes state, and given one input bit = one or zero, there are only two states we could transition to and two possible outputs of the encoder. These possible outputs of the encoder are 00_2 and 11_2.

The metric used is the Hamming distance between the received channel symbol pair and the possible channel symbol pairs. The Hamming distance is computed by simply counting how many bits are different between the received channel symbol pair and the possible channel symbol pairs. The results can only be zero, one, or two. The Hamming distance (or other metric) values computed at each time instant for the paths between the states at the previous time instant and the states at the current time instant are called branch metrics. For the first time instant, the results are saved as "accumulated error metric" values, associated with states. For the second time instant on, the accumulated error metrics will be computed by adding the previous accumulated error metrics to the current branch metrics.

At t = 1, we received 00_2. The only possible channel symbol pairs we could have received are 00_2 and 11_2. The Hamming distance between 00_2 and 00_2 is zero. The Hamming distance between 00_2 and 11_2 is two. Therefore, the branch metric value for the branch from State 00_2 to State 00_2 is zero, and for the branch from State 00_2 to State 10_2 it's two. Since the previous accumulated error metric values are equal to zero, the accumulated metric values for State 00_2 and for State 10_2 are equal to the branch metric values. The accumulated error metric values for the other two states are undefined. Figure-10 illustrates the results at t = 1:

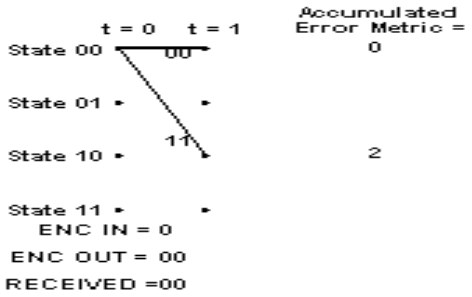

Figure 9: The Viterbi decoding at stage t = 1

Note that the solid lines between states at t = 1 and the state at t = 0 illustrate the predecessor-successor relationship between the states at t = 1 and the state at t = 0 respectively. This information is shown graphically in the figure, but is stored numerically in the actual implementation. To be more specific, or maybe clear is a better word, at each time instant t, we will store the number of the predecessor state that led to each of the current states at t.

Now let's look what happens at t = 2. We received a 11_2 channel symbol pair. The possible channel symbol pairs we could have received in going from t = 1 to t = 2 are 00_2 going from State 00_2 to State 00_2, 11_2 going from State 00_2 to State 10_2, 10_2 going from State 10_2 to State 01_2, and 01_2 going from State 10_2 to State 11_2. The Hamming distance between 00_2 and 11_2 is two, between 11_2 and 11_2 is zero, and between 10_2 or 01_2 and 11_2 is one. Adding these branches metric values to the previous accumulated error metric values associated with each state that came from to get to the current states. At t = 1, we could only be at State 00_2 or State 10_2. The accumulated error metric values associated with those states were 0 and 2 respectively. The figure below shows the calculation of the accumulated error metric associated with each state, at t = 2.

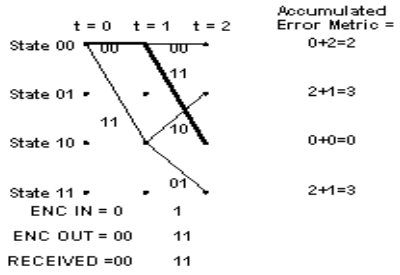

Figure 10: The Viterbi decoding at stage t = 2

That's all the computation for t = 2. What we carry forward to t = 3 will be the accumulated error metrics for each state, and the predecessor states for each of the four states at t = 2, corresponding to the state relationships shown by the solid lines in the illustration of the trellis.

Now look at the figure for t = 3. Things get a bit more complicated here, since there are now two different ways that we could get from each of the four states that were valid at t = 2 to the four states that are valid at t = 3. So how do we handle that? The answer is, we compare the accumulated error metrics associated with each branch, and discard the larger one of each pair of branches leading into a given state. If the members of a pair of accumulated error metrics going into a particular state are equal, we just save that value. The other thing that's affected is the predecessor-successor history we're keeping. For each state, the predecessor that survives is the one with the lower branch metric. If the two accumulated error metrics are equal, some people use a fair coin toss to choose the surviving predecessor state. Others simply pick one of them consistently, i.e. the upper branch or the lower branch. It probably doesn't matter which method you use. The operation of adding the previous accumulated error metrics to the new branch metrics, comparing the results, and selecting the smaller (smallest) accumulated error metric to be retained for the next time instant is called the add-compare-select operation. The figure below shows the results of processing t = 3:

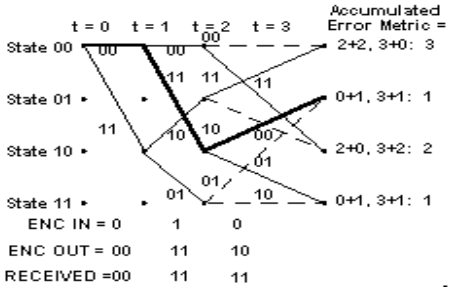

Figure 11: The Viterbi decoding at stage t = 3

Note that the third channel symbol pair we received had a one-symbol error. The smallest accumulated error metric is a one, and there are two of these.

Let's see what happens now at t = 4. The processing is the same as it was for t = 3. The results are shown in the figure-12:

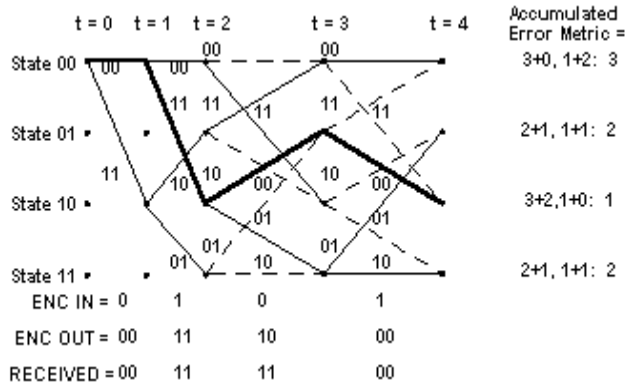

Figure 12: The Viterbi decoding at stage t = 4

Continue this way until t=17, the trellis should look like figure-13, with the clutter of the intermediate state history removed:

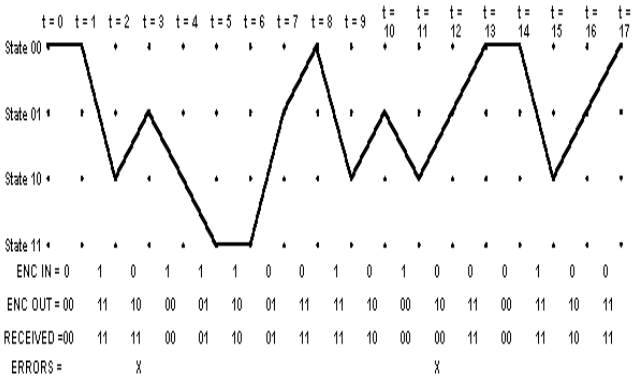

Figure 13: The trellis that reached during the Viterb encoding at stage t =17

The decoding process begins with building the accumulated error metric for some number of received channel symbol pairs, and the history of what states preceded the states at each time instant t with the smallest accumulated error metric. Once this information is built up, the Viterbi decoder is ready to recreate the sequence of bits that were input to the convolutional encoder when the message was encoded for transmission. This is accomplished by the following steps:

First, select the state having the smallest accumulated error metric and save the state number of that state.

Iteratively perform the following step until the beginning of the trellis is reached: Working backward through the state history table, for the selected state, select a new state which is listed in the state history table as being the predecessor to that state. Save the state number of each selected state. This step is called traceback.

Now work forward through the list of selected states saved in the previous steps. Look up what input bit corresponds to a transition from each predecessor state to its successor state. That is the bit that must have been encoded by the convolutional encoder.

The following table shows the accumulated metric for the full 15-bit (plus two flushing bits) example message at each time t:

t=	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**	**12**	**13**	**14**	**15**
St.00 2		0	2	3	3	3	3	4	1	3	4	3	3	2	2	4
St. 01_2			3	1	2	2	3	1	4	4	1	4	2	3	4	4
St. 10_2		2	0	2	1	3	3	4	3	1	4	1	4	3	3	2
t=	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**	**12**	**13**	**14**	**15**

It is interesting to note that for this hard-decision-input Viterbi decoder example, the smallest accumulated error metric in the final state indicates how many channel symbol errors occurred. The following state history table shows the surviving predecessor states for each state at each time t:

t=	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
St.00 2	0	0	0	1	0	1	1	0	1	0	0	1	0	1	0	0
St. 01_2	0	0	2	2	3	3	2	3	3	2	2	3	2	3	2	2
St. 10_2	0	0	0	0	1	1	1	0	1	0	0	1	1	0	1	0
St. 11_2	0	0	2	2	3	2	3	2	3	2	2	3	2	3	2	2

The following table shows the states selected when tracing the path back through the survivor state table shown above:

t =	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**	**12**	**13**	**14**	**15**
	0	0	2	1	2	3	3	1	0	2	1	2	1	0	0	2

Using a table that maps state transitions to the inputs that caused them, we can now recreate the original message. Here is what this table looks like for our example R = 1/2 K = 3 convolutional code:

	Input was, Given Next State =		**Input was, Given Next State =**	
Current State	0	**Current State**	0	**Current State**
00_2 = 0	0	**00_2 = 0**	0	**00_2 = 0**
01_2 = 1	x	**01_2 = 1**	x	**01_2 = 1**

$10_2 = 2$	x	**$10_2 = 2$**	x	**$10_2 = 2$**
$11_2 = 3$	0	**$11_2 = 3$**	0	**$11_2 = 3$**

Note: In the above table, x denotes an impossible transition from one state to another state.

So now we have all the tools required to recreate the original message from the message we received:

t=	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**	**12**	**13**	**14**	**15**
C	0	1	0	1	1	1	0	0	1	0	1	0	0	0	1

Note that the two flushing bits have been discarded.

III. CONCLUSION AND DISCUSSION

This paper presented Convolutional coder software implementation using Viterbi decoding algorithm to decode bitstream that has been encoded using Forward error correction (FEC) to improve the capacity of a channel by adding some carefully designed redundant information to the data being transmitted through the channel. The detailed description and the steps involved in simulating a communication channel - using convolutional encoding with Viterbi decoding - all have been elaborated and discussed in details. The steps involved generating the random binary data, convolutionally encoding the data, passing the encoded data through a noisy channel, quantizing the received channel symbols, and finally performing Viterbi decoding on the quantized channel symbols to recover the original binary data. The steps of modulating the channel symbols onto a transmitted carrier, and then demodulating the received carrier to recover the channel symbols have been left out. This is because we can accurately model the effects of AWGN even though we bypass those steps. Finally, we can say that Convolutional Coding with Viterbi Algorithm decoders has proven itsef as a powerful method that can be rely on and trusted. Therefore, we can see that Convelutional Coding withe Viterbi Algorithm currently used in about one billion cellphones, which is probably the largest number in any application

REFERENCES

[1] Viterbi .J, "Error Bounds for Convolutional Codes and an Asymptotically Optimum Decoding Algorithm," *IEEE Transactions on Information Theory*, Vol.13, April, 1967, pp. 260-269.

[2] [2] Lin, Ming-Bo, "New Path History Management Circuits for Viterbi Decoders," *IEEE Transactions on Communications*, vol. 48, October, 2000, pp. 1605-1608.

[3] [3] V. Pless, Introduction to the Theory of Error-Correcting Codes, 3rd ed. New York: John Wiley & Sons, 1998.

[4] [4] S. B. Wicker, *Error Control Systems for Digital Communication and Storage* . Englewood Cliffs, NJ: Prentice Hall, 1995.

[5] [5] M. S. Roden, *Digital Communication Systems Design*. Englewood Cliffs, NJ: Prentice Hall, 1988.

[6] [6] A. M. Michelson and A. H. Levesque, *Error Control Techniques for Digital Communication*. New York: John Wiley & Sons, 1985.

[7] [7] G. D. Forney, Jr., "Convolutional Codes II: Maximum-Likelihood Decoding," *Information Control*, vol. 25, June, 1974, pp. 222-226.

[8] G. Eason, B. Noble, and I. N. Sneddon, “On certain integrals of.